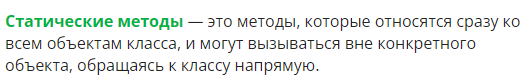
**B7**

**B7.1. Введение**

Похоже в этом модуле будут разбираться дополнительные исключения ООП.

**B7.2**. **Cтатические методы**

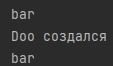


Пока не совсем понятно. В конце прошлого модуля видел описания методов переопределения стандартных операторов такие как сложение, умножение и т.д. Хотя в самом модуле об этом вроде не было слов. **Пойду перепроверю.** Да под магическим методом имелся ввиду только \_\_init\_\_, но они какого-то черта у себя вписали и другие. Ладно. Посмотрим о том ли речь здесь.

Или может этот методы типа **\_\_class\_\_.\_\_name\_\_**

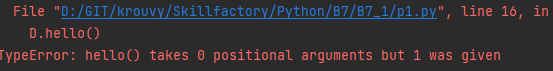
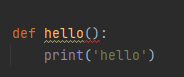
**Если так, то не плохо. Кое что с этим я делал.**

Вот.

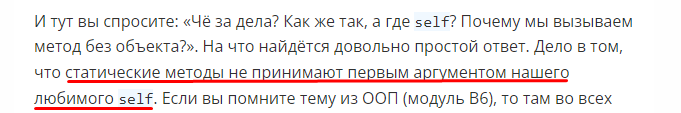


Я сначала подумал, что этот метод доступен для всех классов. Но я ошибся. Суть в том, что такие методы можно вызывать и без создания класса, т.к. он не требует атрибута **self**

Если попытаться без этого декоратора создать метод, то без **self он работать не будет.**

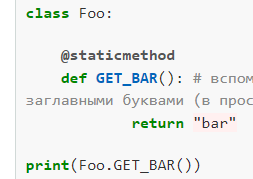
****

А декорированный работает вполне. Такие методы свободно наследуются, разумеется если их передать в дочерний класс.



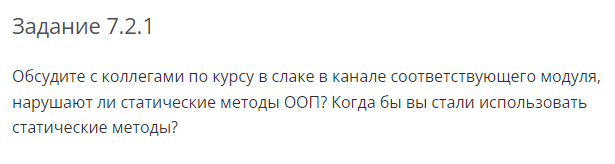
Ладно. Но такие методы никак не зависят от состояния объекта. Так зачем они нужны?

Ну.. советуют в них например хранить константы. ХЗ, как-то не очень.



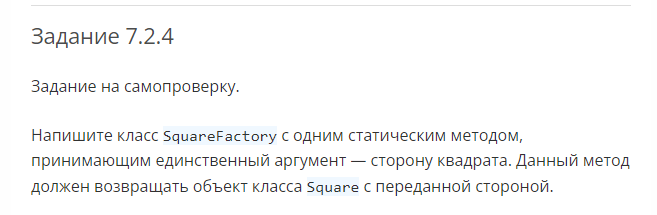
**Используйте статические методы в основном для работы с внешними ресурсами (API, файлы и т. д.).**

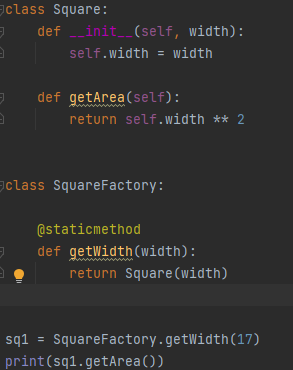
Может в этом метода хотя бы количество атрибутов самого класса можно получить? Эм.. нет нельзя, для этого тоже self нужен. Тогда в таких методах я не вижу смысла. В чем проблема использовать те же функции?



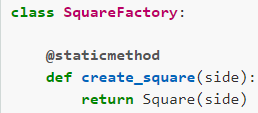
**Вам делать нечего, такую хрень в заданиях вписывать?** Не знаю, как бы я использовал такие методы, но принципов ООП они не нарушают по мне. Как минимум они наследуются.

**Задание плевое, но мне интересно что будет в ответе. Если он дерьмо, то идите нахер.**





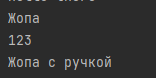
А что написали они?



**Ладно, все хорошо.**

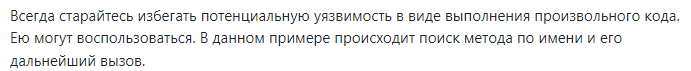
**B7.3. Декораторы класса: @property, @classmethod. Ещё пару слов о нашей бывшей возлюбленной — инкапсуляции.**

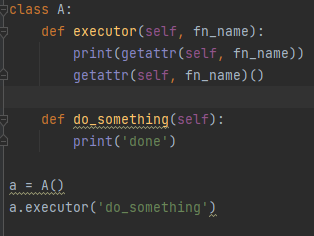
**Так. Внезапно отклонюсь от темы. Помню писал вызов функции по их строковому имени. Вот доп. варианты, которые касаются системных функции самого Python.**

****

**Ниже мне писали, что это писать достаточно опасно, т.к. появляются уязвимости. Но мне просто из интереса.**

Вот вроде негативный пример, но код этот у меня не заработал.







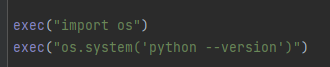
Ну да, работает хотя уязвимость да, создает. Но это просто из интереса.

Через EXEC() Можно даже импорт библиотеки сделать.



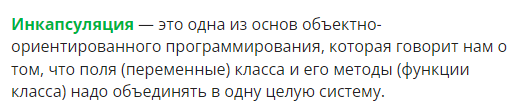
Короче через globals и locals можно найти методы написанные в файле, а через exec и eval, те, что доступны.

В случаи второй строки даже **print** не нужен.

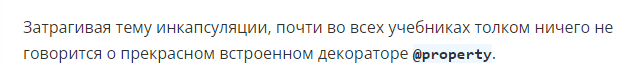
Из-за него лишний ноль в выводе.



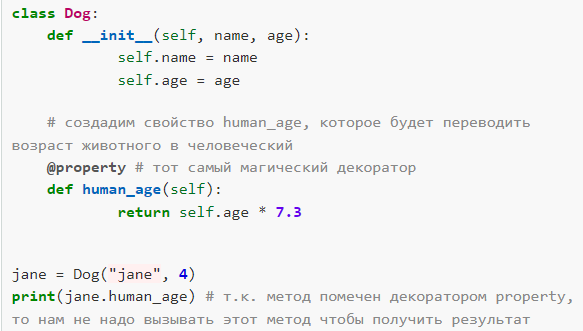
Фух, как же я люблю отвлекаться, когда занимаюсь. Продолжим.



Как всегда, нихера не понятно.



Говорят что **@property** классный декоратор и с помощью него можно объединить поля и методы. Блин, я что с полиморфизмом опять путаю. Вроде нет. Ладно изучим пример.



Понятно. Ну… этот декоратор позволяет вызывать методы таким образом, будто это свойство. То есть без написания скобок(). Сразу видно изъян. Применим только к методам у которых нет атрибутов. А так, ну формально это неплохая штука для создания Геттеров. Хотя. Стоп. Получить name и age в данном случаи можно ведь и просто так?



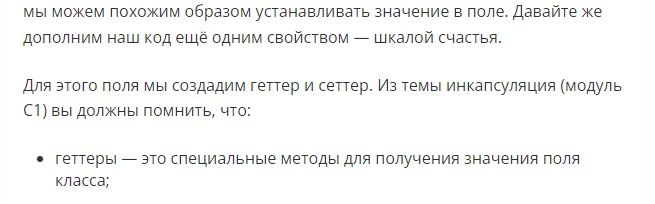
Да. Ну тогда это имеет мало смысла. Лучше бы в тот же \_\_init\_\_ засунуть строку:

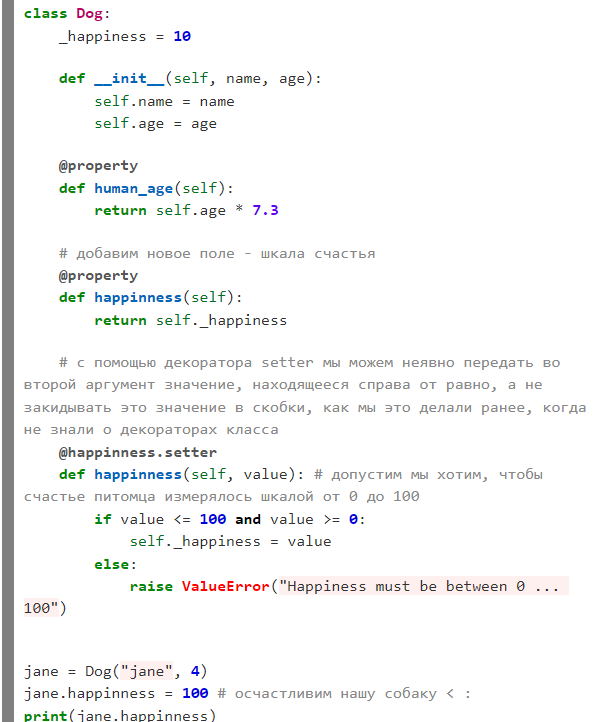
**self. human\_age = age \* 7.3**

Разницы бы не было.

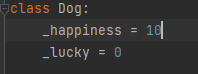
Ладно. Это не все.

**Йоб твою….**

****

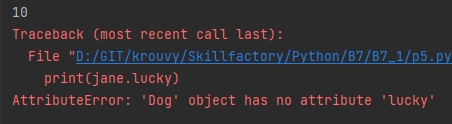
****

**Недостаточно понятно. Во первых теперь да , действительно пошла речь об инкапсуляции. Допустим я добавлю этому классу еще поле Lucky.**

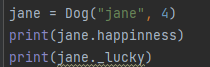
****

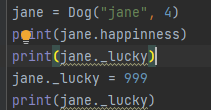
**Но ему я не буду прописывать setter.**

Тогда к этому полю мне вне класса не обратится (без доп. условий).



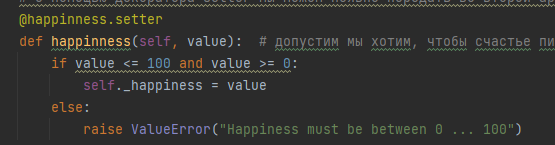
**О, а вот и нихера!**

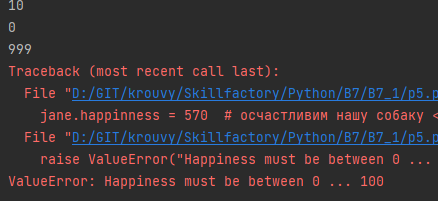
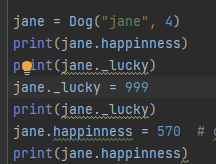




Разница в том, что happiness теряет при обращении из вне свою частичку \_, lucky нет.

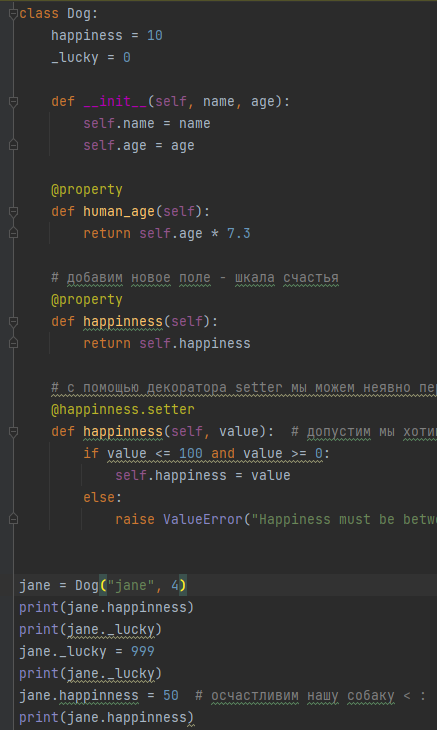
А… тут речь о другом. Тут в декораторе условие, которое похоже проверяет каждый раз, когда значение поля меняется. Если попытаться задать happiness больше ста или меньше нуля вызовется условие и отработает ошибка.

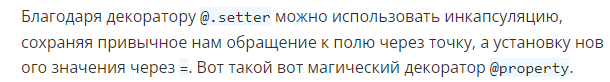




**А нюанс с названием связан с тем, что уже поле \_happiness(**Сука это поле с 3-мя повотряющимеся буквами у вас с ошибкой написано!!!!**),** становится **self\_полем, через @property**

Но тогда мне вообще не понятно, нахера вы в название этой переменной добавляли вот этот символ. Можно же было так оставить!

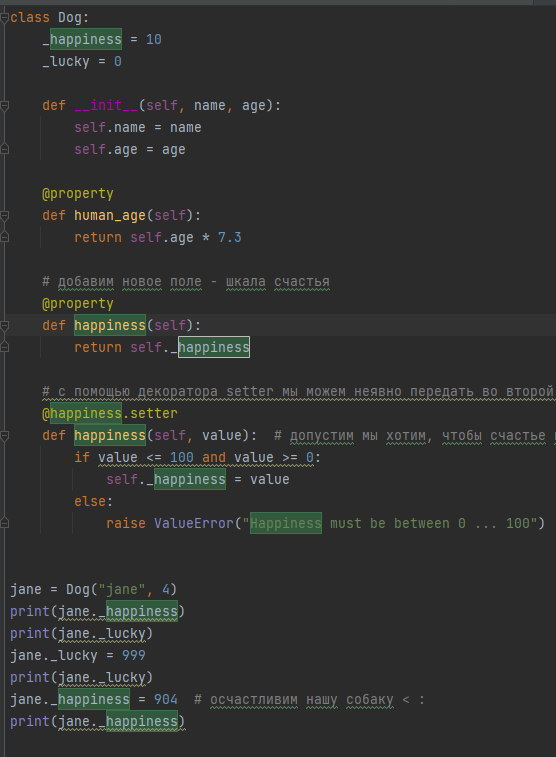




Да, только сама инкапсуляция пропадает, ну хотя не совсем. Таким образом конечно можно добавить защиту и пользователь не сможет напрямую изменить поле на что захочет. Ладно.

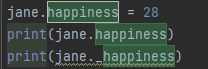
**~~А… теперь все стало на своим места.~~**

**Теперь вообще нихера не понятно!!!**

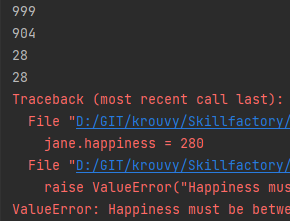
****

**Где теперь моя защита от больше 100? Нигде? Потому что я обращаюсь к полю напрямую через \_имя.**

**А защита не работает потому что я обращаюсь к имени напрямую: \_happiness.**

****

**Поэтому метож декоратора не срабатывает. Чтобы он срабатывал имя должно быть одинаковым с названием метода декоратора.**

****

**Ебать вы пидарасы. Чтобы вся эта хуйня работала, в названии инкапулируемой переменной должно было быть 2 НИЖНИХ ПОДЧЕРКИВАНИЯ, И НЕ ОДНО!!!!**

****

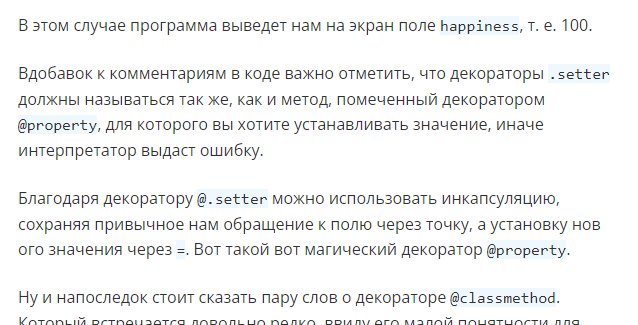
**Оставляя таким образом к этому поля нормально не поступится из вне да через:**

****

Теперь эту переменную можно изменить только через декоратор. И в декораторе есть свое доп. условие, что тоже хорошо. Таким образом инкапуляция есть.

И почему это я должен объяснять себе, а не вы мне!!!!!

Пойду посмотрю, что там пишут.



Нихуясе, пиздец пидарасы. Вот поэтому по вашим курсам говнопрограммисты, которые нихера не понимают и появляются. Нихуя по коду не объяснили и пошли дальше. Пиздец.

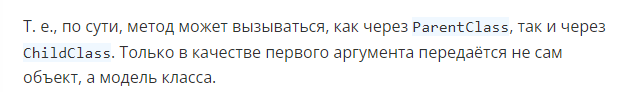
**А штука то полезная, но действительно чтобы догнать, надо было подумать.**

**Остыл. Можно продолжать.**

**@classmethod**

**Пишут что он мало понятен и громоздок и из-за этого используется не часто. Ну надеюсь в этот раз вы объясните получше.**

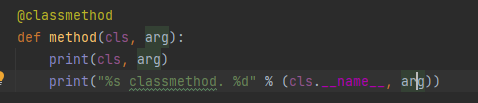
****

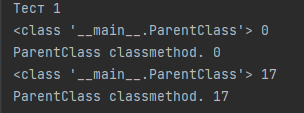
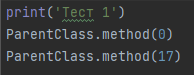
****

**Кода сразу не мало. А объяснений нихуя Еб вашу мать.**

**Придется опять самому разбобрать.**

**Добавил доп поля на всякий случай. И 2 вызова.**

****

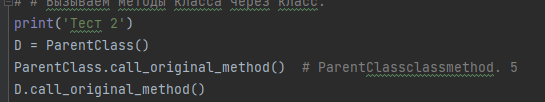
****

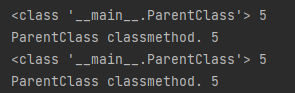
**Похоже декоратор позволяет получить информацию о самом классе. У всех декораторов, появляется аргумент cls.**

**Через cls.\_\_name\_\_ к примеру получают имя класса. Только тут без предупреждения используется % строка, не помню как она называет, даже забыл слегка. Но она как f строка, только сама находит место для вставки.**

**Ладно нижние методы:**

****

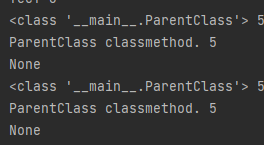
****

****

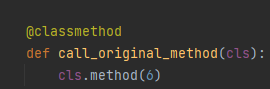
**Эм… не понял.**

**Ладно, может print() нужен?**

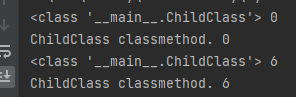
**print ничего не дал.**

****

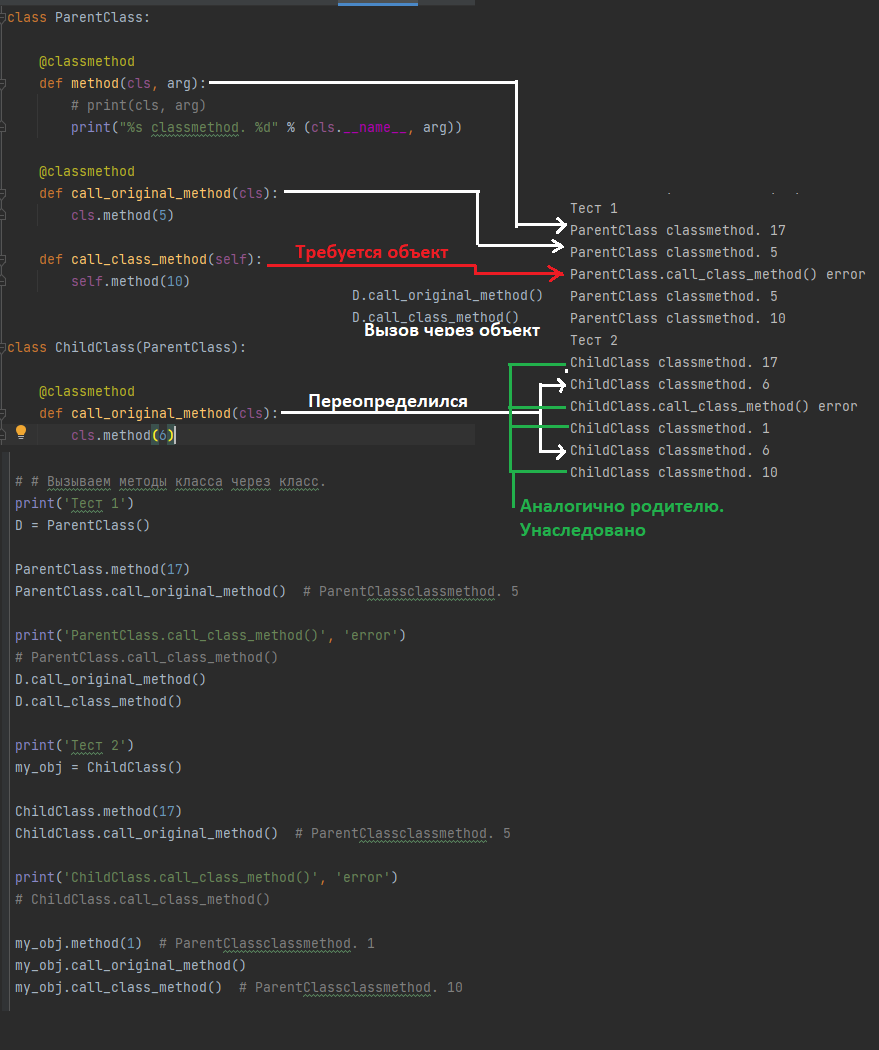
**Все время вызывается метод, у которого в выводе 5. Как получить 10. Через дочерний класс?**

****

****

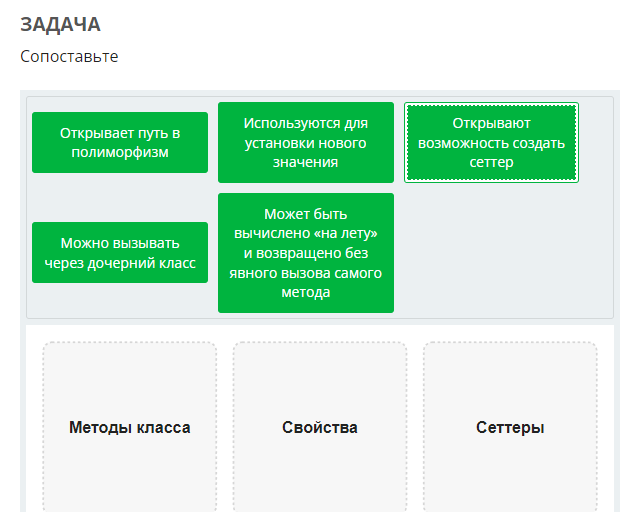
****

**Так понял. Моя ошибка в тесте 2. Я там не то название написал.**

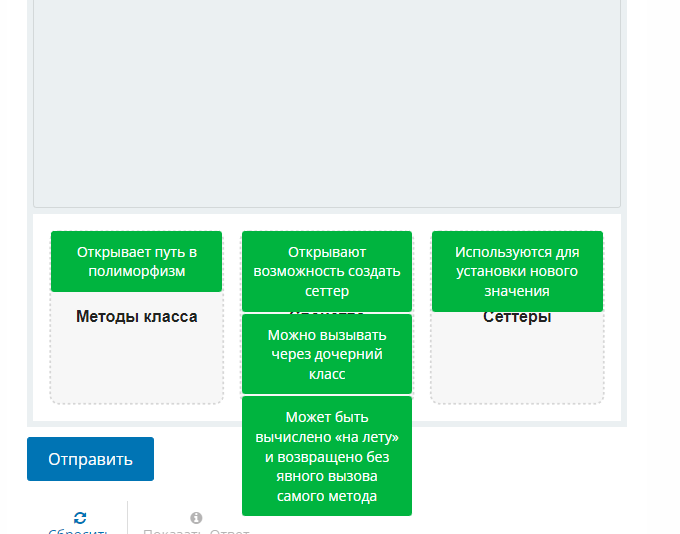
****

**Так. А теперь к теме. Нахера еще раз он нужен? Но с помощью него можно получить инфромацию о текущем классе, даже без объекта. Это не плохо. Конец.**

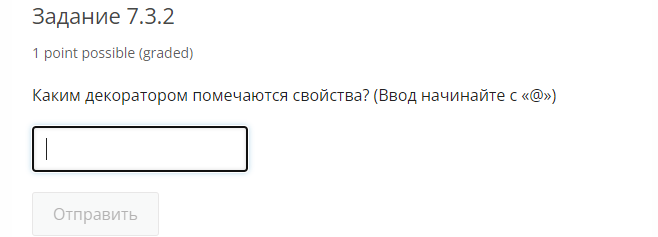
**А… мои мозги.**

****

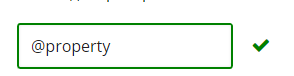
**Еще одна неочевидная хуета сделанная наотъебись. У вас сайт лагает, а вы чему-то учить пытаетесь.**

****

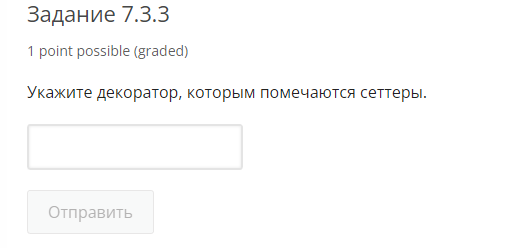
**Круто да… “Открывает возможно создать сеттер” Конечно это Свойство.**

****

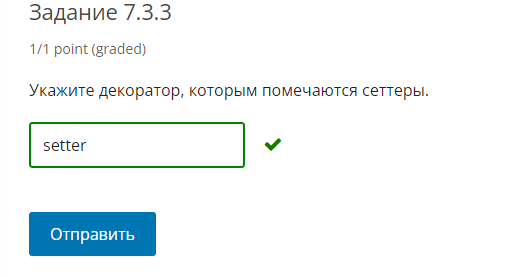
**Что? Я не помню.. вроде только методы же разбирали…**

****

**Ладно. Но блять, им не свойства помечают, а методы сетторы. Которые, да, устанавливают значения свойств.**

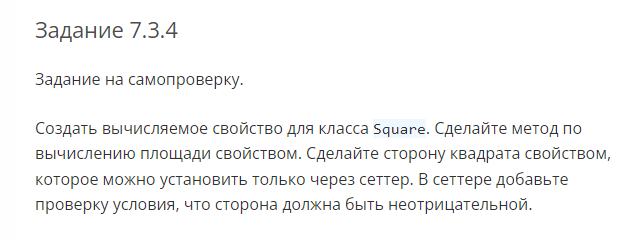
****

**Да еб вашу…**

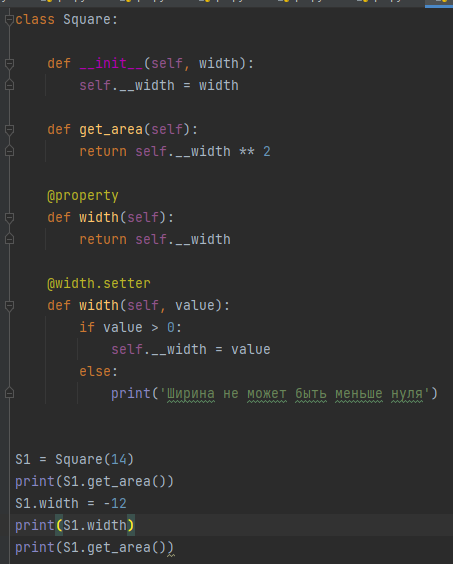
****

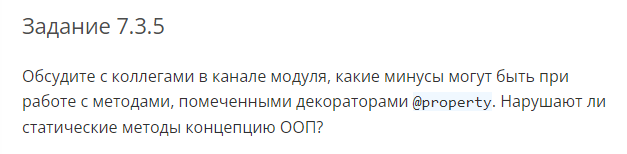
**Задавайте вопросы корректнее.**

**P.S. Пока бомбил названия правда забыл. Так что за возможность закрепления спасибо.**

****

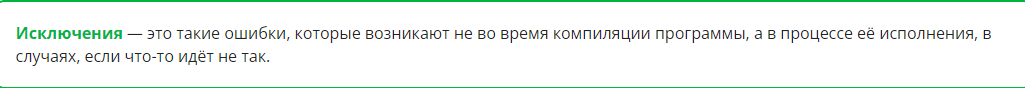
Сделал через пару дней, не зря. Одно дело просто изучать код, но переписав усвоил его лучше. Да и пока писа, пару раз подглядывать пришлось так как

****

****

Если использовать просто голый @property, без setter, то минус в том, что он не имеет возможности принимать аргументы из вне, так как отрабатывает сразу, как идет обращение. Не помню что там за статические методы, но с помощью него и сеттора выходит отличная инкапсуляция.

**B7.4. Исключения**

****

Ошибки бывают отлавливаемые и не отлавливаемые.

К неотлавливаемым относят **SystemExit, KeyboadrdInterrupt** и т.д.

Отлавливаемые – все, что наследуется от класса Exception. Исключения пишутся и для доп. библиотек могут писаться свои.

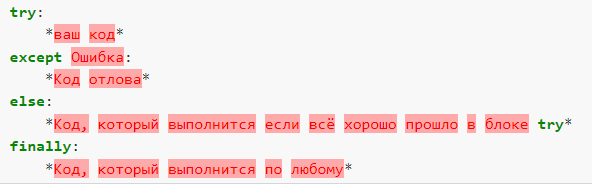
Вот примеры:



Это лишь часть, все вроде как здесь: <https://docs.python.org/3/library/exceptions.html>

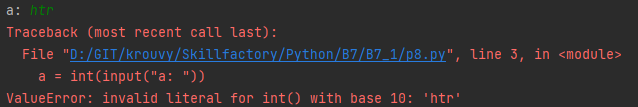
При каждом возникновении исключения программа будет останавливаться. Чтобы этого избежать нужно использовать конструкцию **try – except.**

В **try** нужно поместить опасный код, а в **except** описать возникающую ошибку.

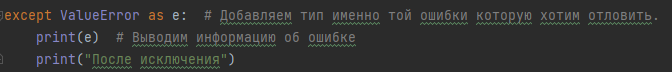


Сейчас покажу пример. Допустим я хочу строку: ‘htr’ преобразовать в тип данных int().

Получу ошибку.

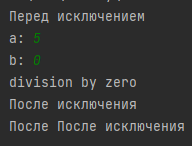
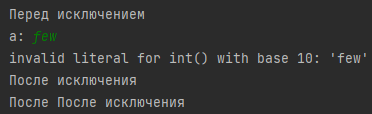
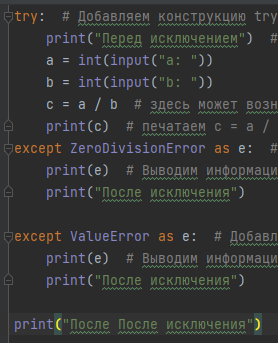


Добавлю в код обработчик исключения **ValueError**.

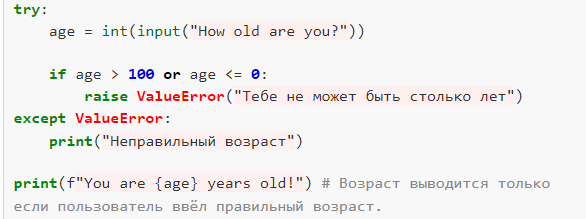


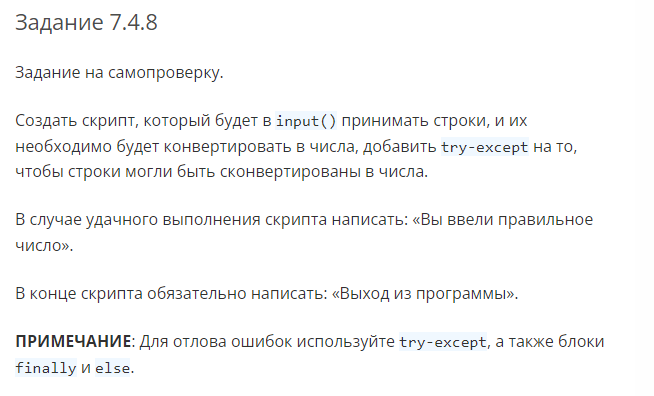


Вот код целиком. Деление 2-х чисел



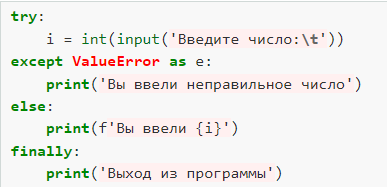
Через **try except** можно обрабатывать в том числе и собственные исключения, написанные через **raise**.





Делать не буду, потому что примерно это уже выше добавил. **ValueError**

Ну вот ответ от них.



тут все элементарно, так что добавить нечего.

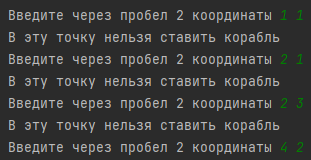
**B7.5. Итоговое практическое задание**

У.. звучит как что-то серьезное. Ну давай, попробуй. Это было задание: морской бой. Писал его 3 дня. Идеальной я бы программу не назвал. 555 строк. 3 Класса, 4 метода, 10 функций(некоторые из которых корректно было бы сделать методами, ну в прочем ладно).

Функционал нормален. Компьютер рисует поля корректно, есть защита от случаев, когда компьютеру проблематично нарисовать корабли в маленьком пространстве. Сначала защита до счетчика в 70 раз, а затем еще 3 таких раза. Если за эти 210 попыток не нарисует, то прервет программу.



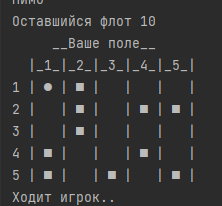
Что касается игрока на него это тоже распространяется. Но сделано не удобно. Вот ситуация: уменьшенное поле 5 на 5. Но игрок расположил на поле корабли так, что 4-ый корабля в одну клетку просто негде больше установить:



В таком случае игроку придется 70 раз пытаться поставить корабль в любой из этих точек.

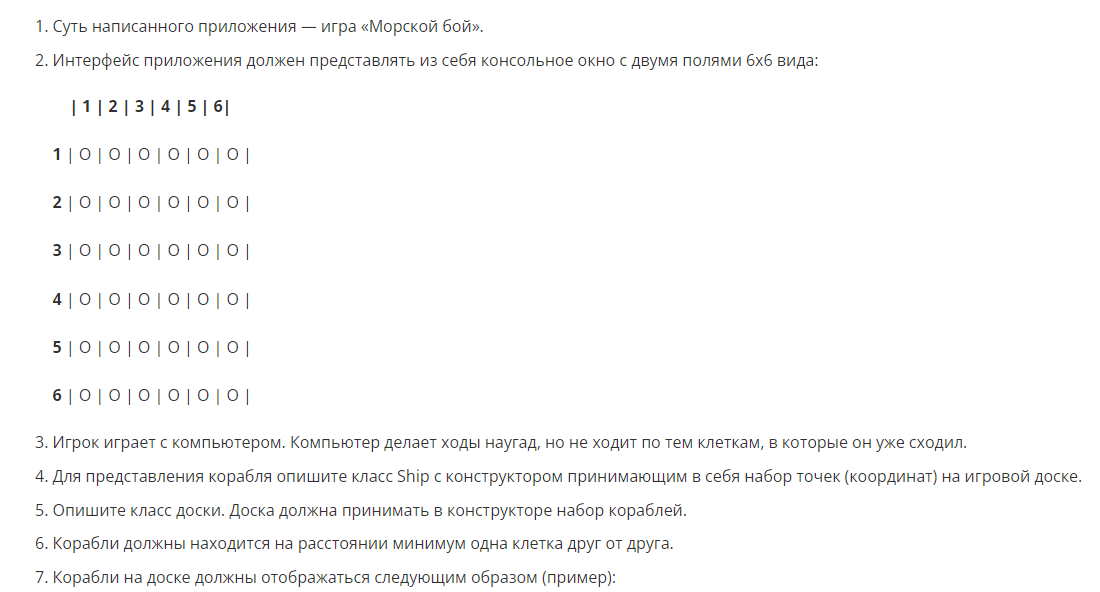


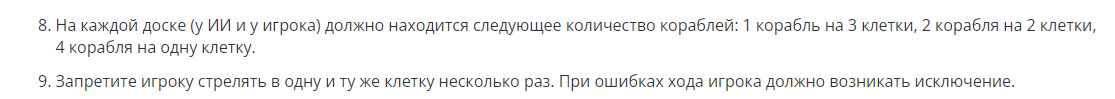
В этом случае игра не прервется, но у игрока будет на один корабль меньше.

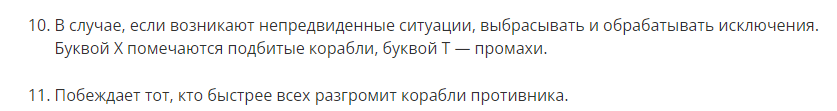


Дважды по одним и тем же полям не позволено стрелять ни игроку не компьютеру. Можно регулировать размер игрового поля. По хорошему можно было бы еще добавить и регулировку количества кораблей в зависимости от размера поля, но думаю хватит. Надо будет посмотреть вариант от самого курса. Попробовать его сломать в разных местах и сравнить размер кода.

Вот таким правилам я должен был следовать. Рассмотрим их.







**2. Соблюдено**

****

**3. Соблюдено,** по возможности можно доработать тем, что компьютер заменяется другим игроком, каждое поле будет отображаться именем игрока, а реальные поля не будут отображаться для обоих игроков(в данный момент игрок видит на поле все свои корабли, когда по ним стреляют.)

**4.Соблюдено частично.** У меня есть отдельный класс для корабля, который принимает в качестве аргумента не координаты, а длину корабля. Но координаты корабля у этого класса имеются, но задаются они позже.

**5. Не соблюдено.** Класса для доски не описывал, тем не менее в зависимости от параметра длины игровое поле может изменятся У меня есть класс **GameProccess,** где для игрока и противника создаются 2 флота со своим набором кораблей. И через метод **GameProccess** игровые поля и заполняются кораблями. Вот именно! Игровые поля. У компьютера и игрока они свои. И изначально одинаковые.

**6. Соблюдено**

**7. Соблюдено.**

**8. Соблюдено, но позже.** Думал нужно всего 3 малых корабля, изменил 4.

**9. Соблюдено частично.** Если игрок стреляет в одну и ту же клетку, программа просто не дает игровому процессу продолжить, пока игрока не введет корректную точку. Обработка исключения у меня происходит только в случае если вместо координат будет введена строка с символами.

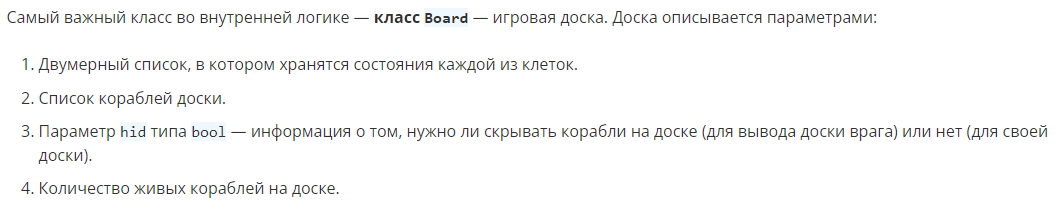
**10. Соблюдено.** Единственным непредвидимым обстоятельством может быть рисование кораблей на уменьшенном поле. На поле 5 иногда не все корабли могут сразу поместится. А на поле 4 вообще не возможно расположить все корабли, с таким полем программа завершится преждевременно.

**11. Соблюдено.**

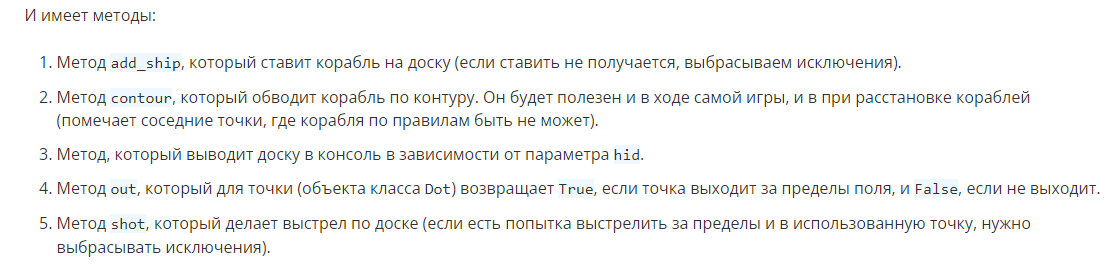
**Вот так предлагается описать класс корабля.**



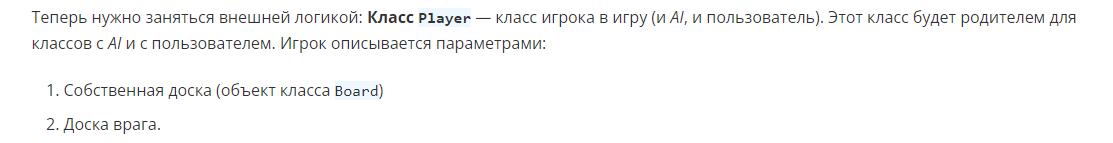
Ну тоже не плохо.



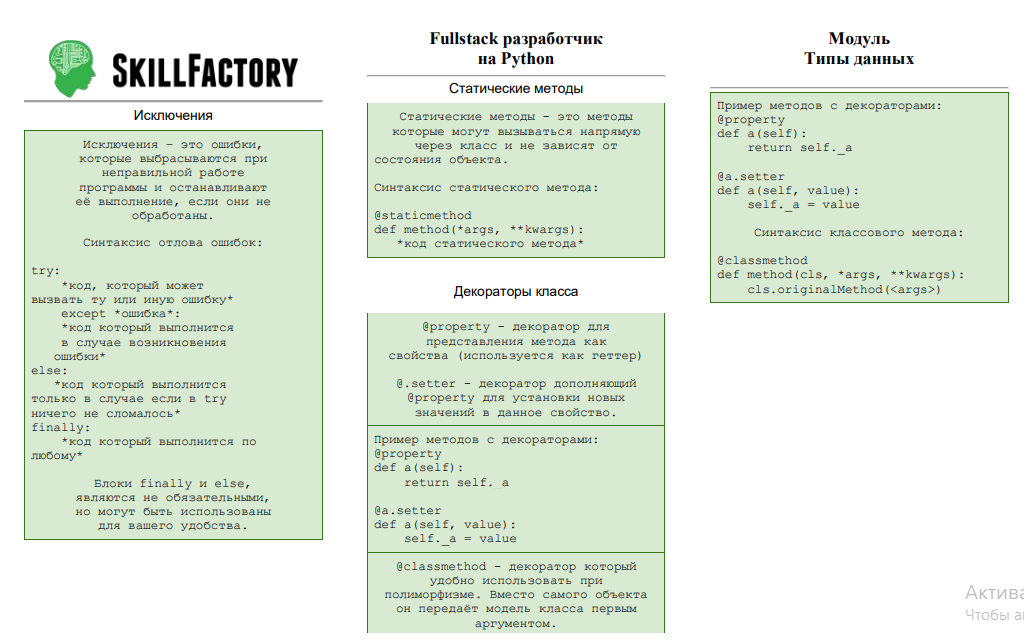
Тут это откровенно лучше чем у меня. У меня не состояния поля, а действительно 2 поля. Одно настоящее с кораблями, а второе просто видимое.



Расстановка схожа. Метод contur, интересная мысль, мне в голову не пришла.

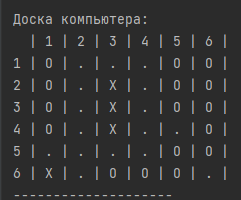


У меня опять же просто 2 Флота, но у каждого флота тоже есть по 2 доски, но это свои доски, не доски врага.



В следующих модулях планируется изучение неких**: новую структуру в Python — контекстные менеджеры.**

**Код себе скачал. На первый взгляд он лучше. Он продуманнее и короче. Что понравилось. Тут корабли действительно имеют жизни. И когда идет попадание по кораблю и у него не остается хп, то вокруг него появляется контур, куда нет смысла стрелять.**

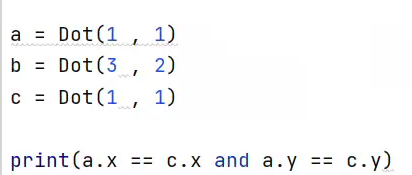
****

Код лучше разберу позже вместе с вебинаром. Сейчас голова устала.

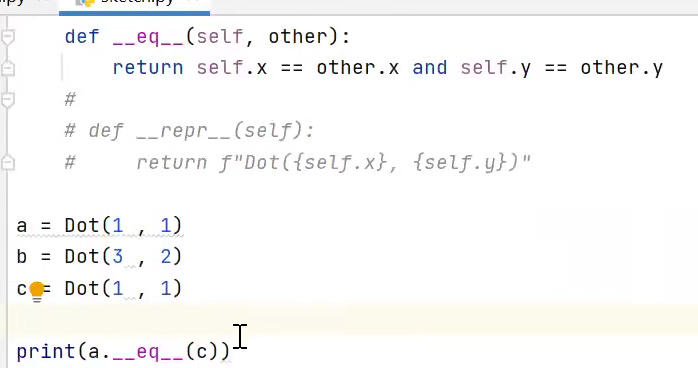
Ладно. Начну разбирать тот вебинар. Чисто сам код одному разбирать слегка скучно и утомительно.

Пример метода, как можно сравнить два равны ли два объекта класса.

Допустим класс, который принимает 2 точки. Хотим сравнить 2 класса у которых точки одинаковые. Чтобы не писать такой код:



Можно реализовать такой метод:



И что удобно в таком методе, так это то, что он реализован через ключевое слово **\_\_eq\_\_**

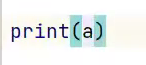
А это дает возможность вызывать этот метод таким образом:



Когда мы просто вызываем объект класса, например кодом:

print(a). То чаще всего мы получаем инфу <class имя класса и тд>

А что если хочется получить что-то конкретное, например у этого класса при обращении к нему видеть информацию о 2 его точках. Реализовать это несложно. Для этого нужно написать функцию: **\_\_repr\_\_** и вернуть ей ту информацию, которую хотим. Пример.



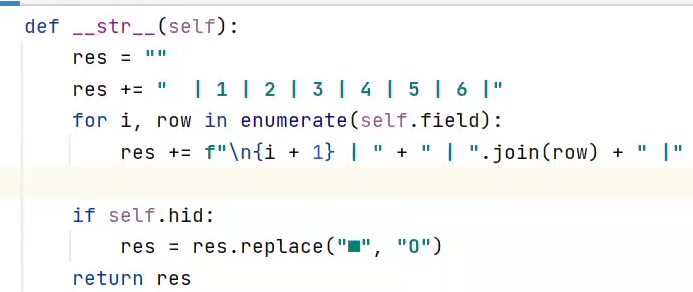


Я не знал про это. Удобно.

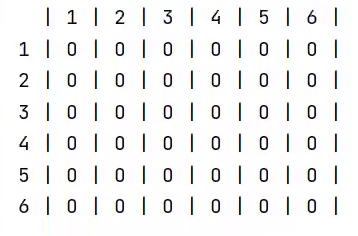
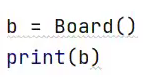
Класс корабля описан так. Передается стартовая точка, и направление вертикальное или горизонтальное, а также длина корабля. И затем относительно этих параметров с помощью инкремента, увеличивается координата по **x** или **y.** К полю пока никак не привязано. И ничего не сказано, что если стартовая точка будет например 5, 5 для поля длинной 6 и корабля в 3 клетки. Будет выход за поле.

Не совсем понял, но есть ключевое слово для методов **‘\_\_str\_\_’.**

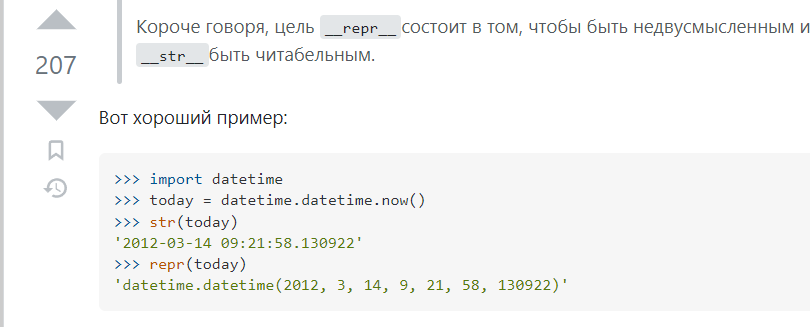
И он похож на **\_\_repr\_\_**



Ну вот тут в функции вывод сформированной строки. И метод вызывается точно также, вот он внутри **print.**

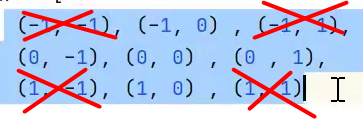
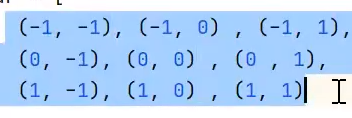


Что говорит интернет на этот вопрос? Разницы у них мало, а простыней текста написано дохера. Чаще всего это сводится к следующему. **REPR для разработчиков и информации, как она есть. А STR для удобного отображения, для пользователей.**



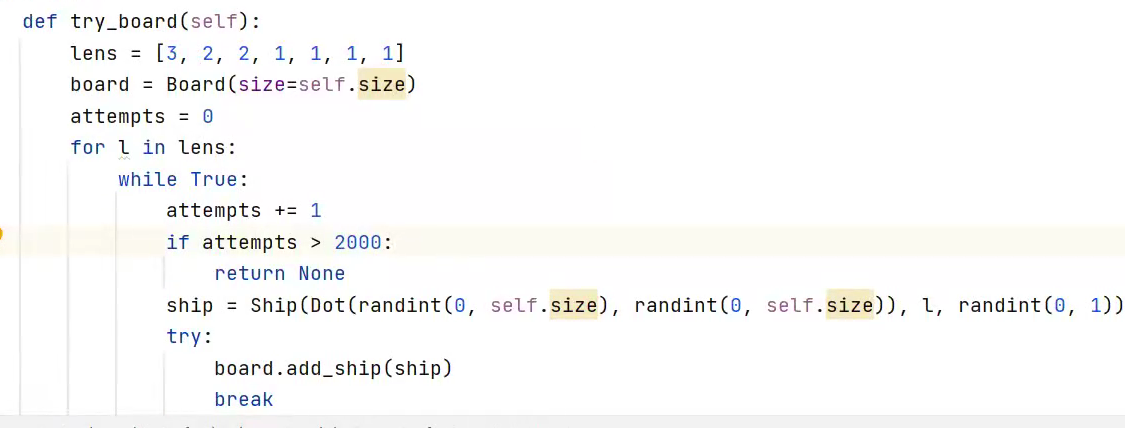
Хех. С их рандномным рисованием досок, у них иногда уходит очень много времени, чтобы нарисовать корабли, у меня в алгоритме если это возможно, то произойдет быстрее. Еще у них при изменении параметра размера игрового поля рисование происходит с ошибкой. Как минимум шапка поля у них всегда будет: “| 1 | 2 | 3 | 4 | 5 | 6 |”.

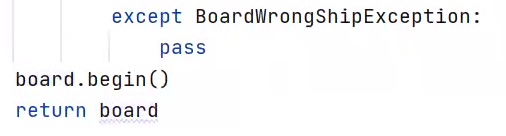
Я должен быть непредвзят, поэтому предположу, что у меня логика еще не совсем верная в плане, где можно ставить корабли. По моей задумке, корабли не могут касаться друг-друга напрямую по стенкам, но углами по диагонали могут. У них же в проге это запрещено. У них точки от корабля следующие. А у меня такие



Все же мое рандомное рисование корабля лучше. Потому что, он относительно всего поля сразу видит все варианты установки и просто выбирает один из рандомных. Если список возможных вариантов пустой, то тогда уже он будет выбирать другую первую точку.

Случайное рисование корабля устроено так:





**lens** = список длин кораблей.

И затем в бесконечном цикле пытается создаться корабль. Все остальное случайно. Единственное, точка проверяется через исключения. И когда все закончится, начнется игра.

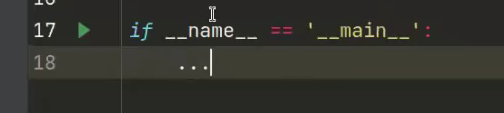
О еще тут в коде есть параметр attempts. У меня было нечто похожее, у меня если попытки превышали 70 раз, то он попытается нарисовать доску полностью заново. И так 3 раза.

Еще можно похвалить код, за то, что здесь можно легко задать длину кораблю например 4, и он все равно сможет такой корабль нарисовать. У меня функция говно и формально для каждой длины свое условие.

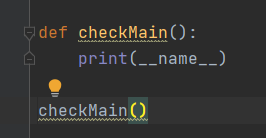
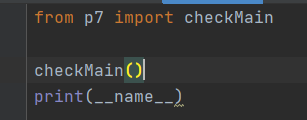
И все же хорошая реализация.

Следующий вебинар: **Практикум по ООП.**

Займусь им в следующий раз.

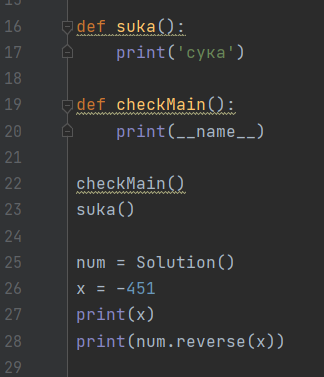


атрибут name будет возвращать \_\_main\_\_ только если этот код был запущен в файле, в котором он написан. Он не будет возвращать main, если был импортирован.

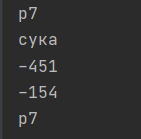




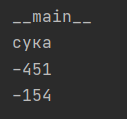
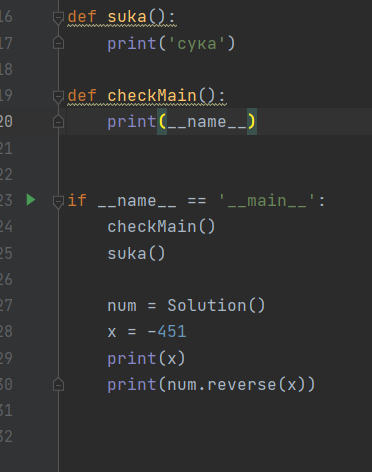
А еще при таком импорте важен нюанс. в p7 у меня есть такой код. А в p8 такой:

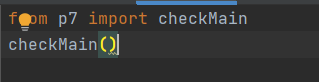


При запуске p8 будет следующий результат:



Оказывается при импорте из файла, если в импортированном файле было прописано какое-то выполнение, то при импорте оно произойдет. Первые 4 строки вывода это файл p7, и только 5-ая строка выполнение файла p8. **Важно отметить то, что выполнение не означает инициализацию.** Функцию Suka() нельзя вызвать внутри p8 т.к. она неимпортирована. Она выводится только из-за того что вызывается внутри p8. Чтобы код не вызывался можно как раз внутри p7 прописать условие с \_\_name\_\_ == main

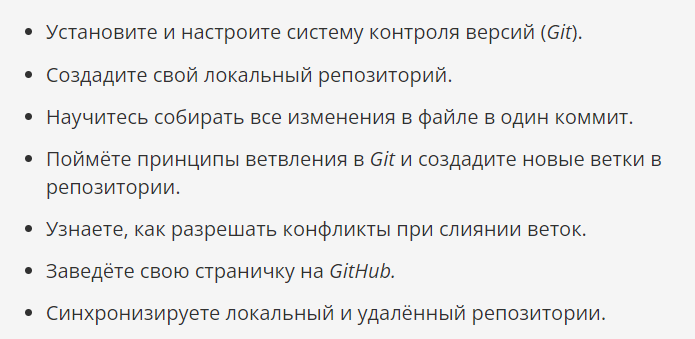




А так почти ничего не узнал в этом вебинаре. Времени чуть жаль.

**Git.GitHub**

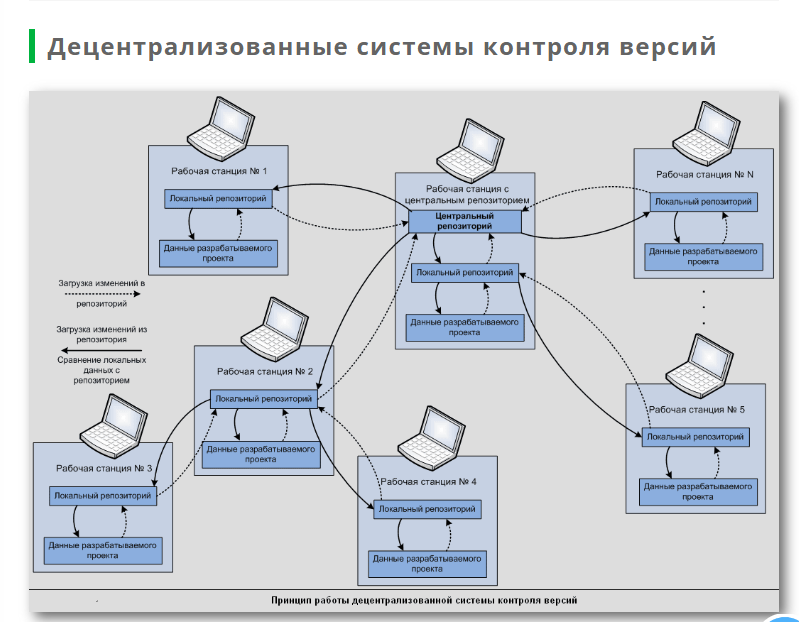
Похоже этот модуль впихнули позже, но в принципе без разницы. Работаю с ним наверно года 1.5 но очень много не знаю.



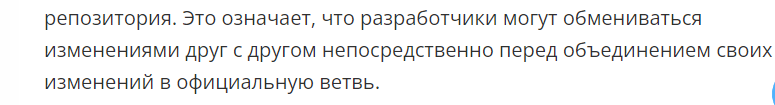
Создавать ничего не буду, наверно… но нижние пункты точно нужно осмыслить. Системы контроля версий бывают 3-х видов. Локальные, централизованные, децентрализованные.

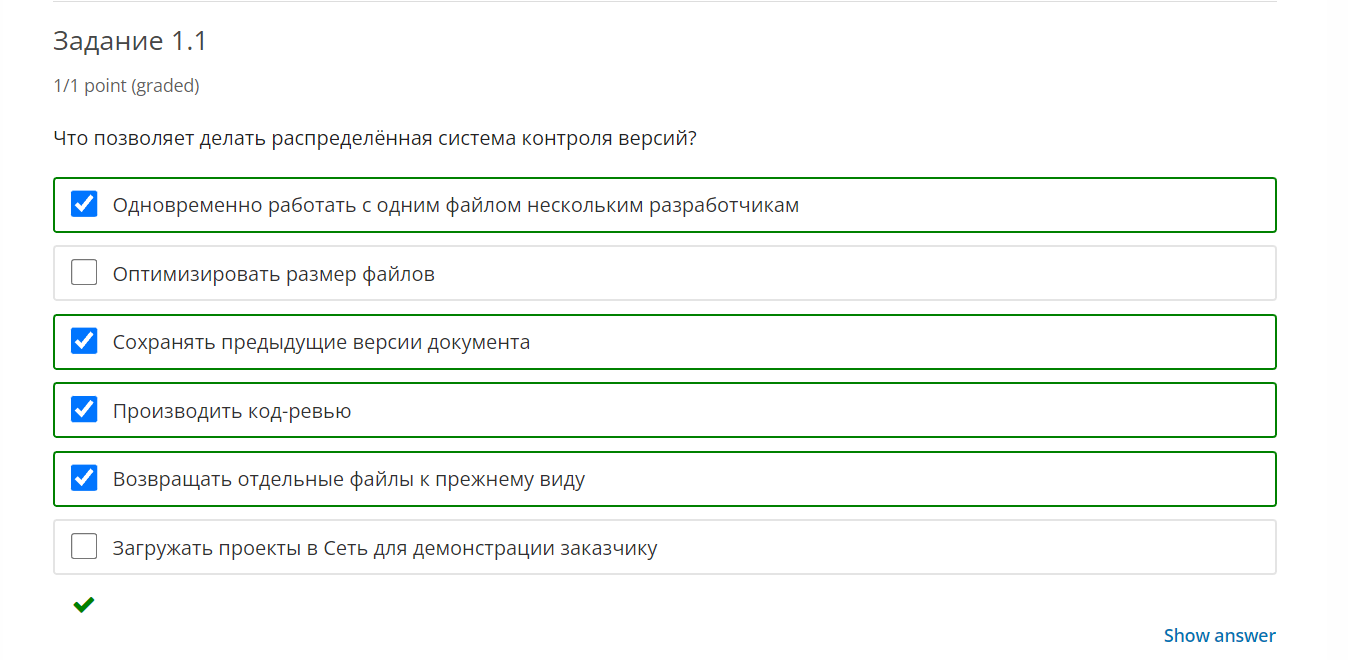


Из этой картинки можно сделать вывод, что github централизованная система?

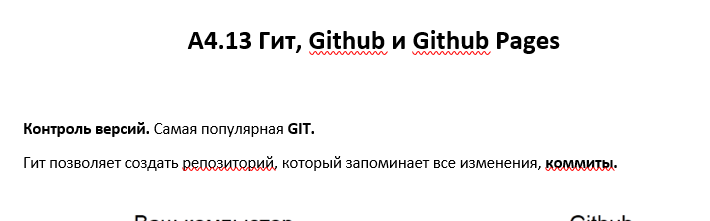


Не совсем очевидно. А, похоже тут как раз речь о GIT.



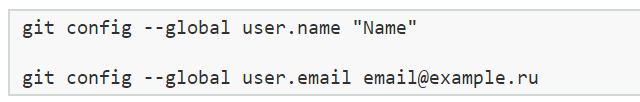


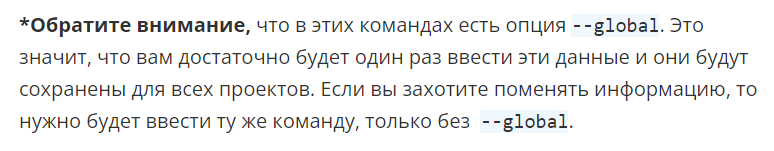
Чмошники с тупым заданием у вас тут вариантов ответ писец. И хер поймешь какой может лишний нажал. В к примеру последний вариант вполне возможен, жопышники. короче. Блин.. я же уже что-то писал про git. Сам изучал? Ну ка гляну.



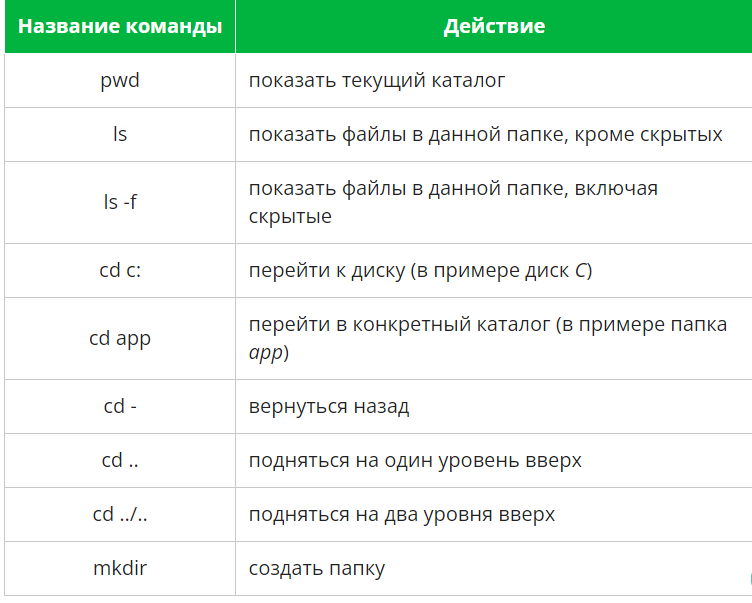
Ага.. Ну ладно, повторение мать учения. Блин. В будущем надо будет создать один файл, в который вставлю все doc файлы. Будет проще искать инфу.

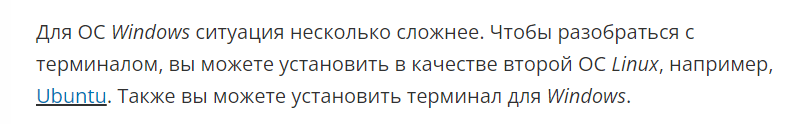
В новом установленном гите сначала принято задать имя и почту.



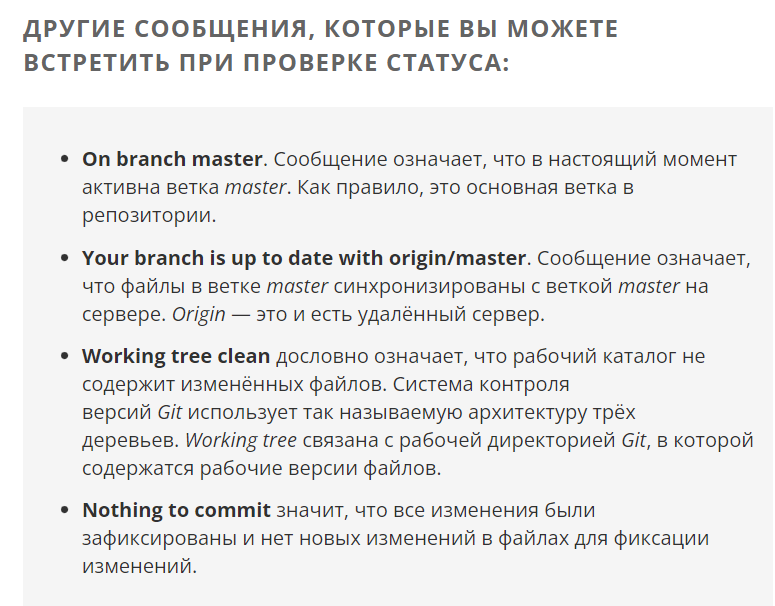


Полезные команды в терминале unix систем. Windows в том числе





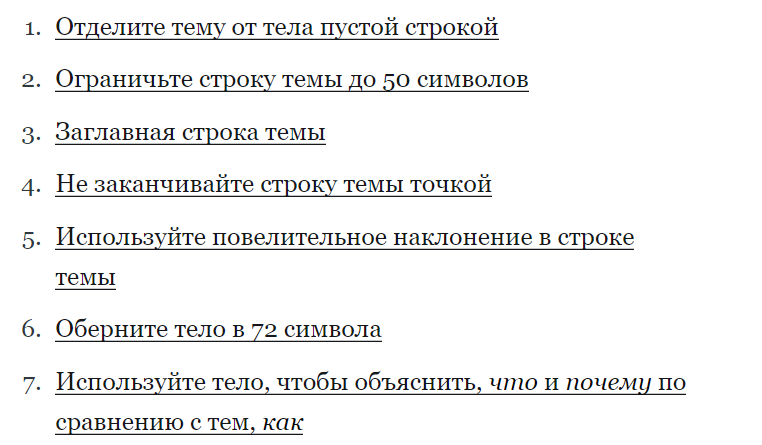
Что значит ОС windows несколько сложнее, вы только что команды выше описали. Тут речь имено о LINUX OS, тупые.



Коммит – это совокупность изменений, сделанная одним разработчиком и сохраненная в определенный момент времени.

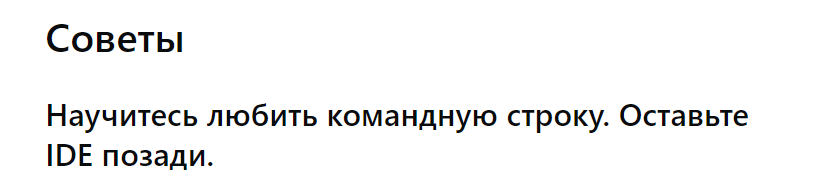
<https://cbea.ms/git-commit/>

Общие правила написания сообщений коммитов.

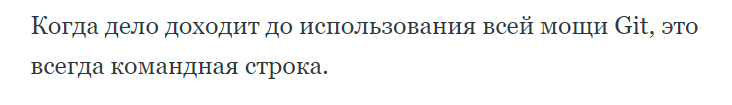


Я бы перевел как: Будь краток. Но и длинные коммиты имеют право быть.

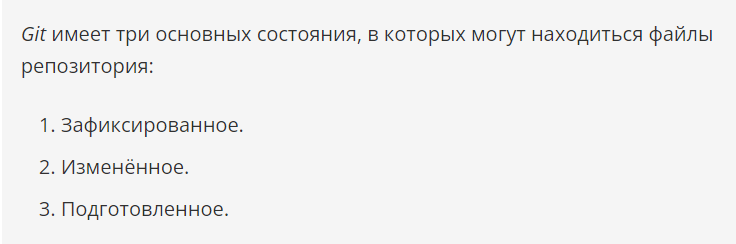


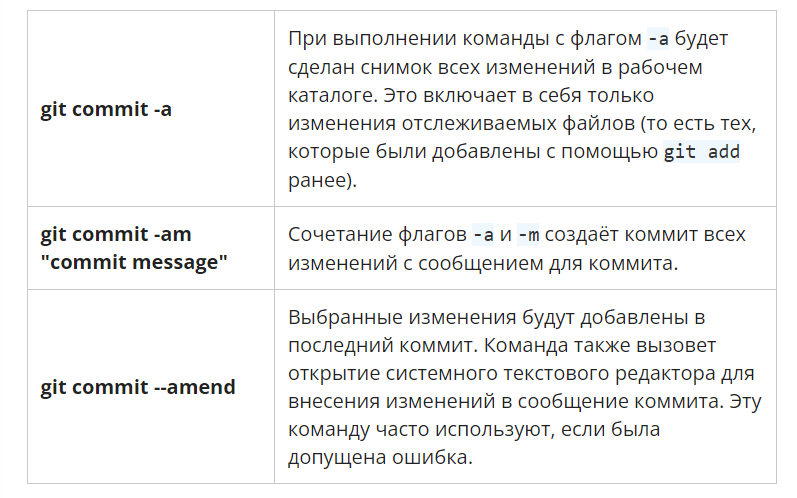


Scrool you!



Может быть в будущем. Когда дойдет до этого.





**Ветвление и коммиты слияния**

В каждом репозитории есть минимум одна ветка. По умолчанию это **master.** Допустим я хочу отдельно написать в текущем проекте одну фичу. Но не затрагивать пока не закончу остальное. Для этого можно создать отдельную ветку.

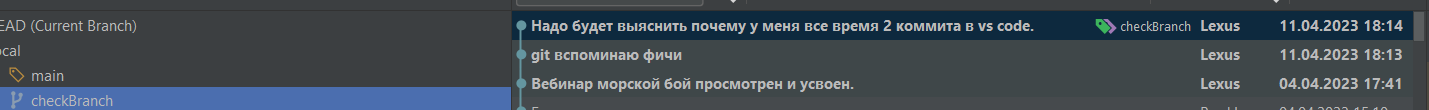
Ветка это всего лишь указатель на коммит, не меняющий репозиторий. Когда делаешь коммиты не создавая и не меняя ветку, то работа проходит под мастером.

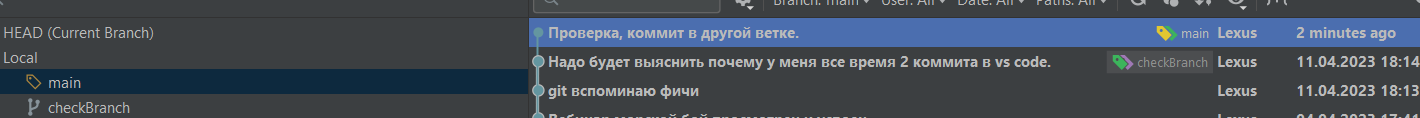
Чтобы создать новую ветку можно прописать

**git branch awesome-branch**

Почему бы мне сейчас не создать ветку с этим текстом.

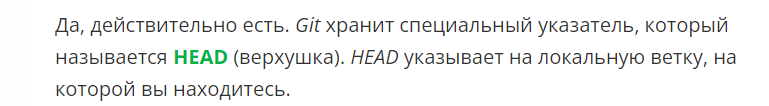
Ветка создалась, но пуш я сделал в другой ветке

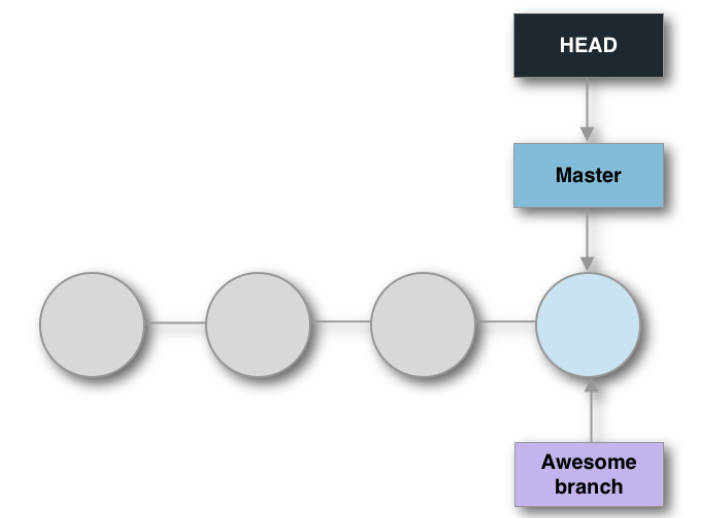




Попробую то, что написал сейчас в ветке на тест залить.

По прежнему на main…





Надо сделать так, чтобы этот head указывал на новую ветку**. Вот команда для этого:**

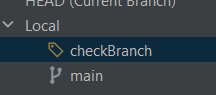
**git checkout \_имя\_ветки\_**

Да это работает. У меня уже есть версия этого файла без текста выше.

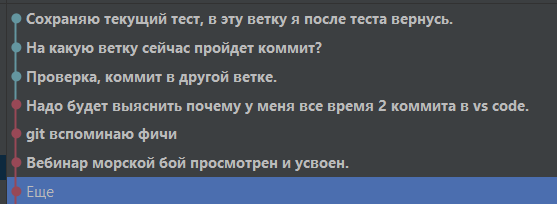
Через IDE это делать даже проще.



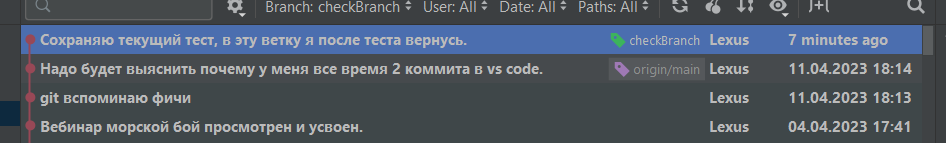
И выбранная ветка сразу перемещается наверх.



Вот как коммиты выглядят на основной ветки. Красная это ветки checkBranch, и у нее не видно ее последнего коммита.

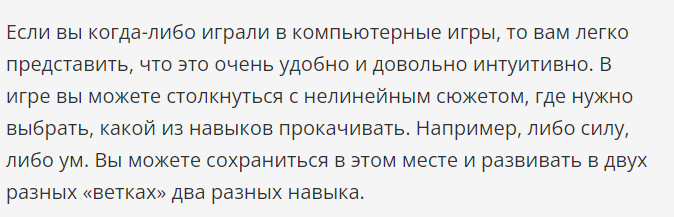


Если на нее переключится, то ветка будет выглядеть так:



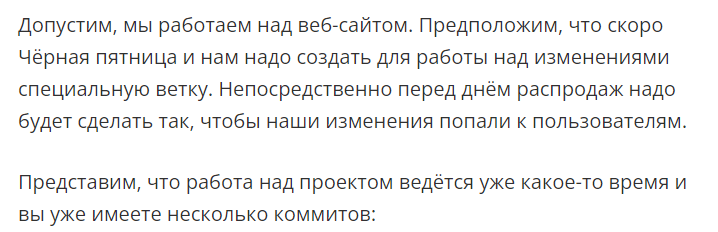
Если делать через терминал, то будет следующее сообщение.





## ****Ветвление и коммиты слияния****

Вот это интересно. Как правильно.

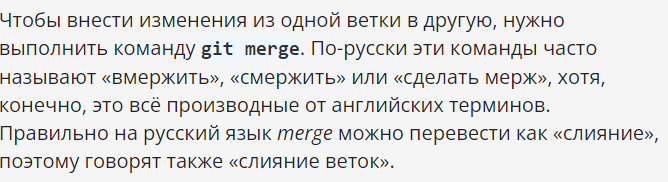


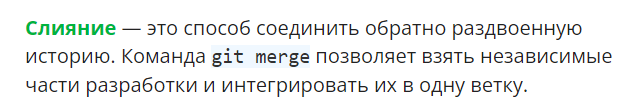
Чтобы начать работу нужно создать эту ветку:

Кстати, если в названии ветки допущено название его можно спокойно изменить:

**git branch -m старое-имя-ветки новое-имя-ветки.**

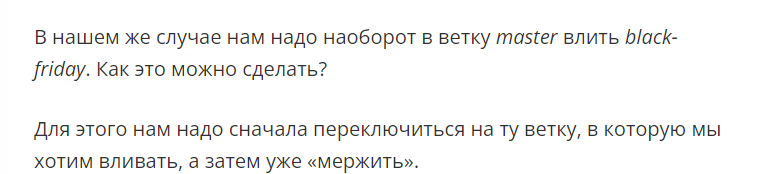
Допустим работа завершена и эти изменения нужно отправить главную ветку.

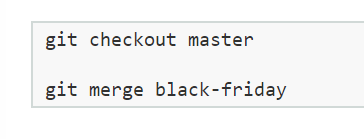




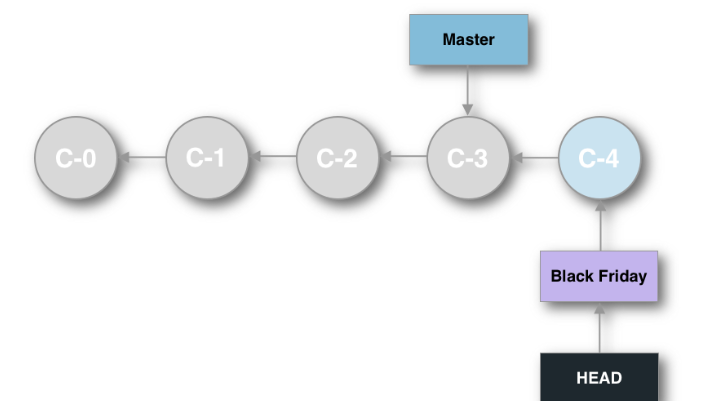
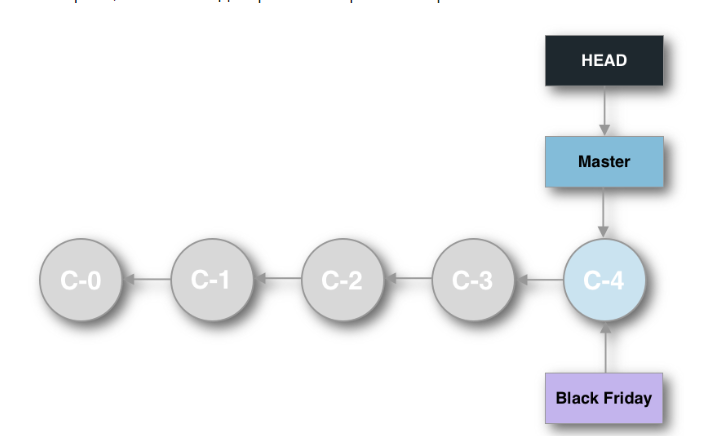
Если бы мы хотели влить новые изменения в текущую ветку (Black Friday), то надо было бы выполнить команду: **git merge master**, что означало бы, что в текущую ветку мы хотим влить ветку с именем master.

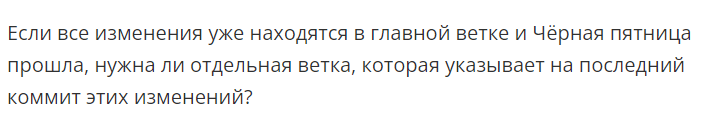
**Однако**





**Было - Стало**



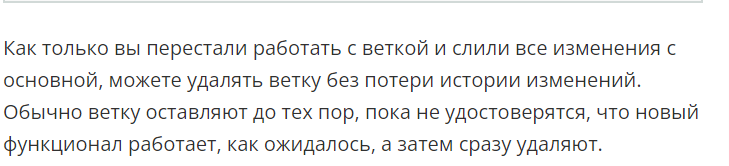
Если в ветке ничего не разрабатывается больше, то она просто будет продолжать указывать на один и тот же коммит, который со времененм будет в далекой истории, и это вроде нормально.

**Если вам когда-нибудь понадобится вернуться к этой версии или посмотреть, какие изменения были тогда внесены, для этого будет достаточно воспользоваться механизмом коммитов.**

Хранить ветку бессмысленно, ее можно удалить. Вот команда.

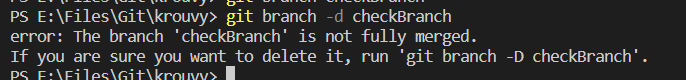
**git branch -d black-friday**

Ну не долго у меня та ветка жила, попробую.



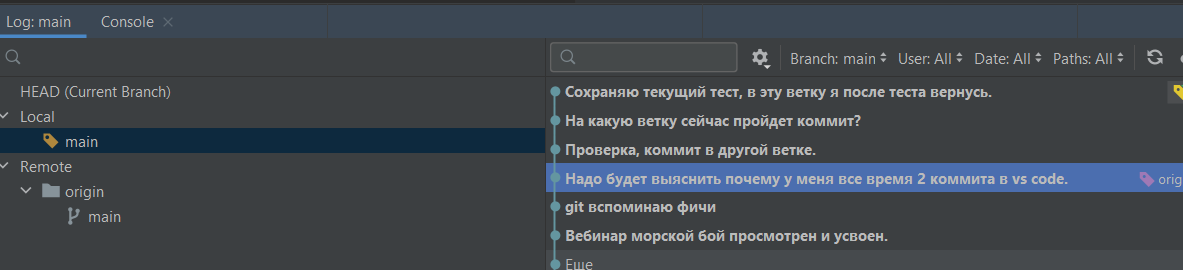
Ну у меня нет необходимости использовать слияние. Поэтому можно спокойно удалить ветку.

Правда так как я не делал merge вышла ошибка, и для удаления нужно использовать большую –D.

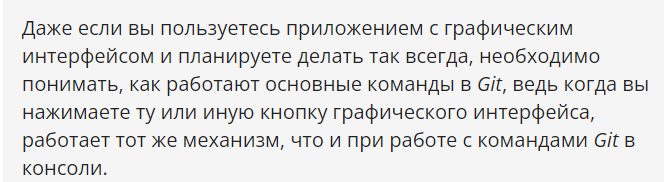




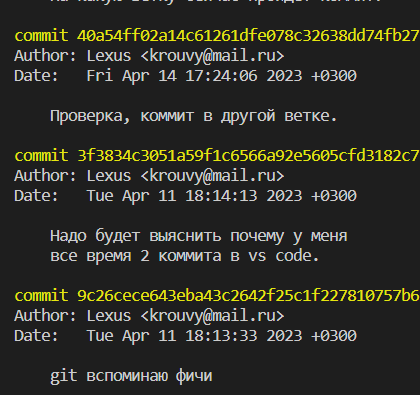
Посмотрим что теперь в IDE

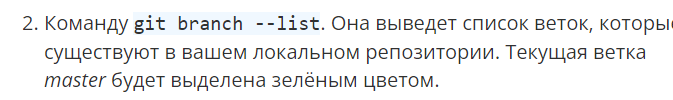


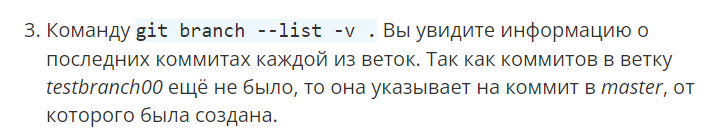
Пустота, но если бы был merge испория бы сохранилась или типо того.

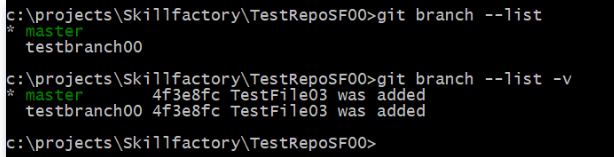


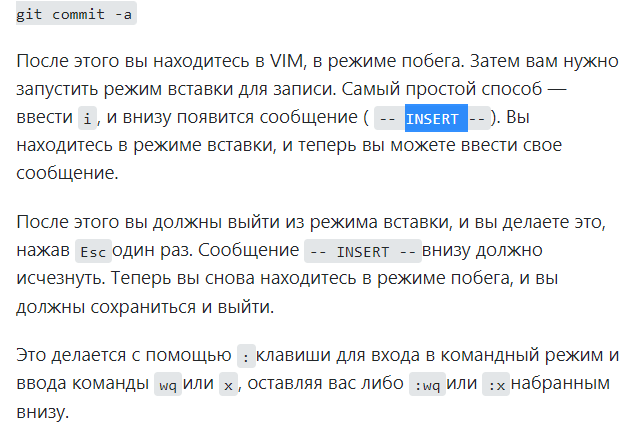
Команда git log показывает историю коммитов. Чтобы выйти из просмотра нужно нажать на клавишу **q.**





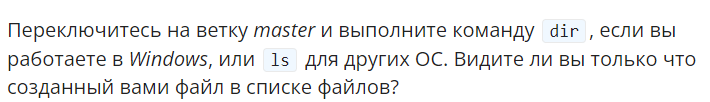




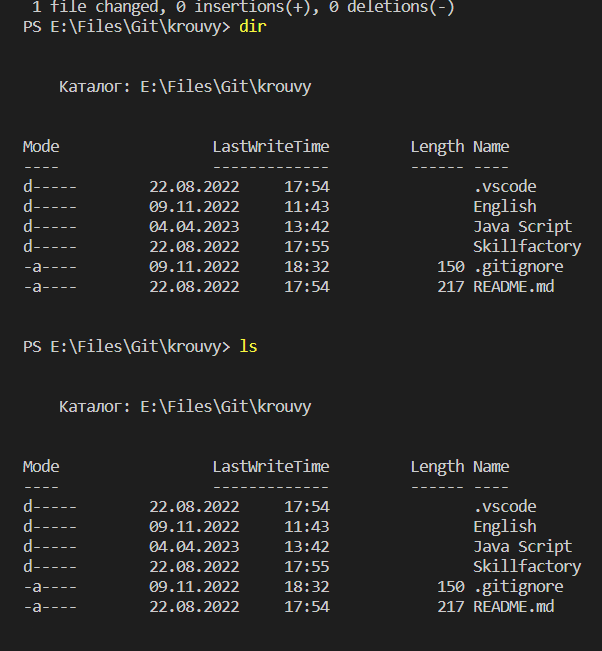


Это если забыть написать сообщение и откроется VIM. Выйти из него можно нажав клавиву : и добавить :qa!

Или как написано выше чтобы сохранить.



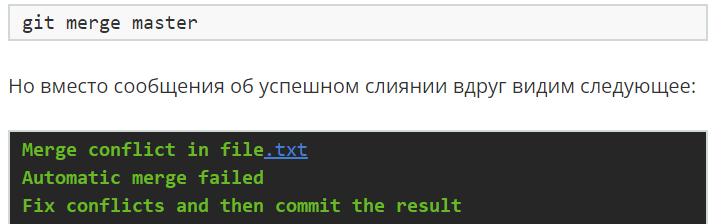
На винде кстати сейчас работает и dir и ls.

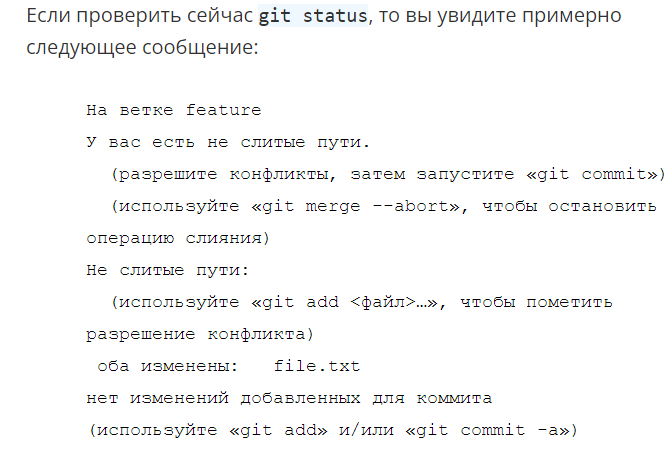


**Конфликты**

На сегодня все. Но хвалю. Год работаю, а по гиту не знаю нихера. Надо будет продолить. А пока сделаю коммит через терминал)

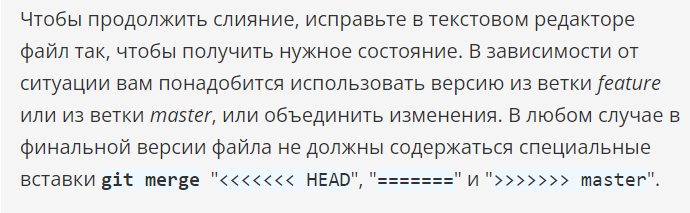
Допустим мы хотим закинуть изменения из главной ветки в дополнительную. Делаем мердж, но получаем ошибку:

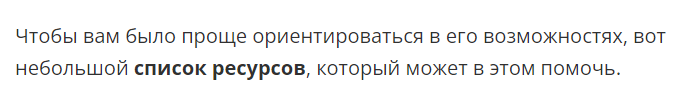




На этом этапе есть 2 пути, первый это отменить слияние командой:

**git merge –abort.**





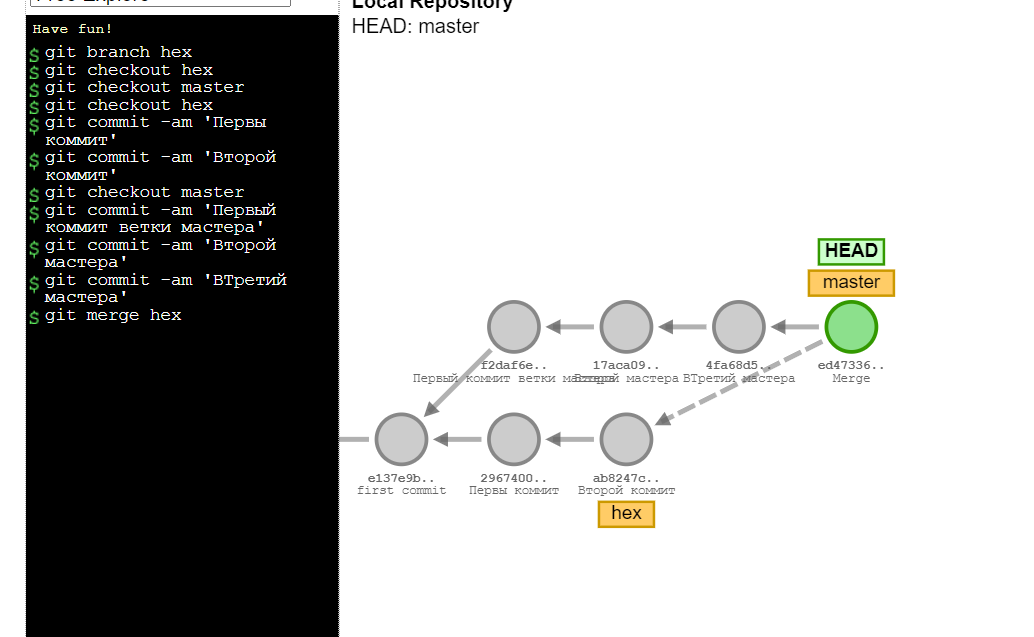
(Нам лень объяснять изучай сам)

Книга по git <https://git-scm.com/book/ru/v2>

Шпаргалка <https://training.github.com/downloads/ru/github-git-cheat-sheet/>

А тут можно типо поиграться

<http://git-school.github.io/visualizing-git/>

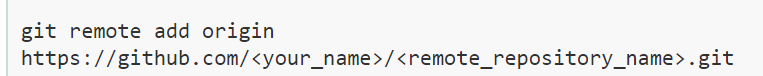


Позже найду время и позанимаюсь в этом тренажере.

<https://learngitbranching.js.org/?locale=ru_RU>

**GitHub**

Это веб сервис куда можно заливать локальные репозитории. Допустим есть созданный репозиторий на GitHub. Его нужно связать с файлами, локального репозетория на компьютере. Вот как это можно сделать.



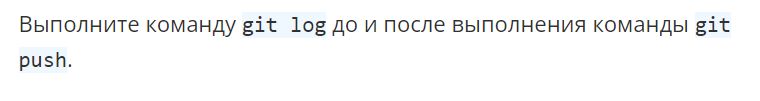
И последним шагом нужно будет сделать пуш этих локальных изменений на удаленный сервер.

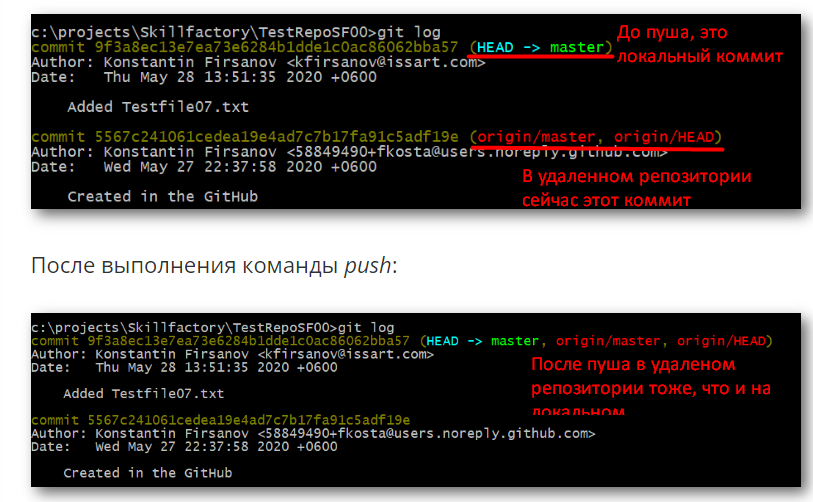
**git push origin master**

Это можно сделать и в обратном порядке, то есть уже есть удаленный репозиторий с файлами. Осталось только на компьютер его клонировать.

**git clone**

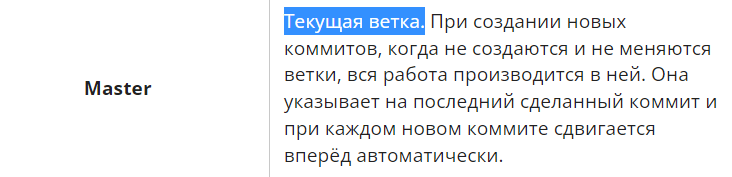
клонировать



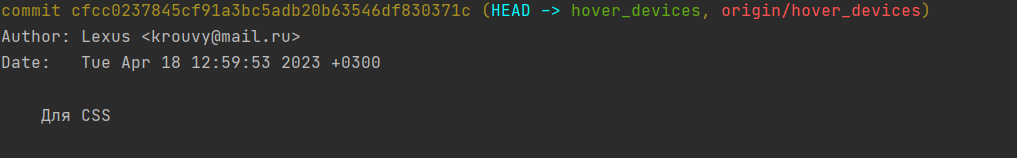


Гайд по гит хабу.

<https://help.github.com/en#dotcom>



Как бы да, но опять же, только если нет других веток. head -> master если в этой ветке, но если работаем в другой, то

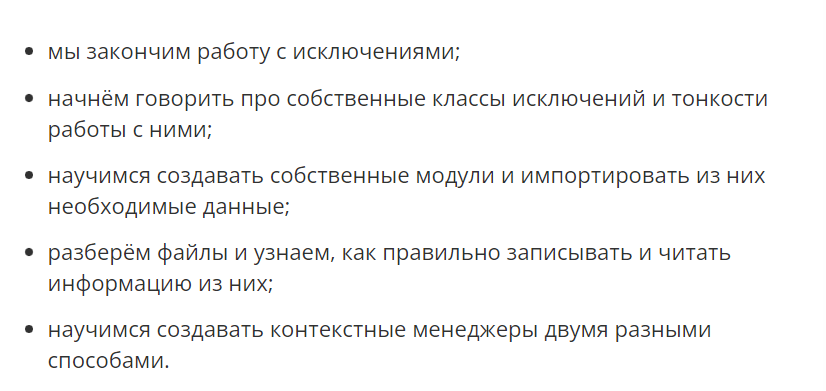




**Модуль B8. Модули и импорт. Работа с файлами и данными.**

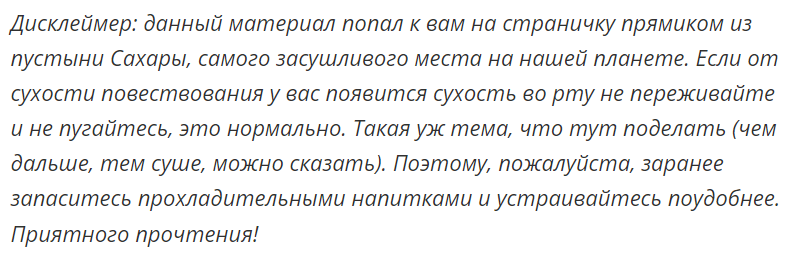
Ладно, вернемся к Python. С гитом конечно надо будет еще по экспериментировать.

Планы.



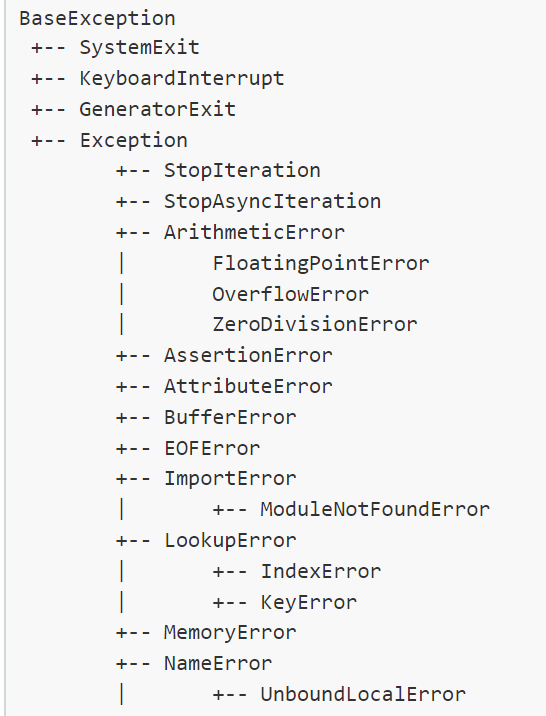
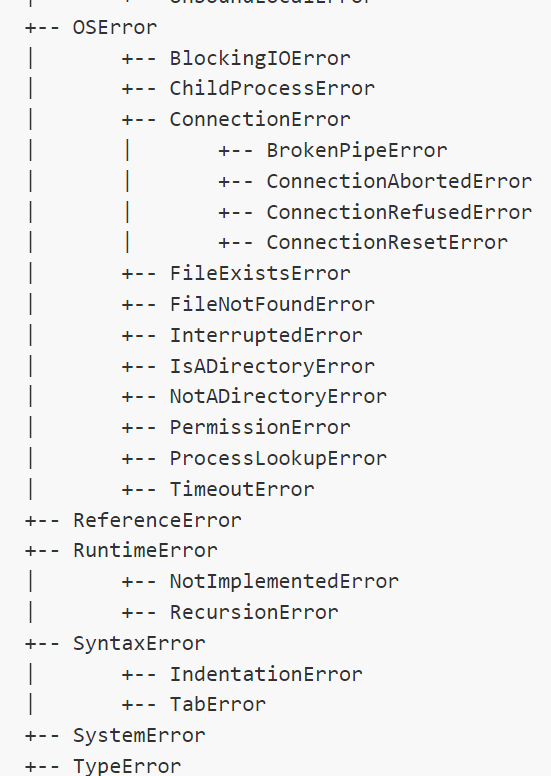
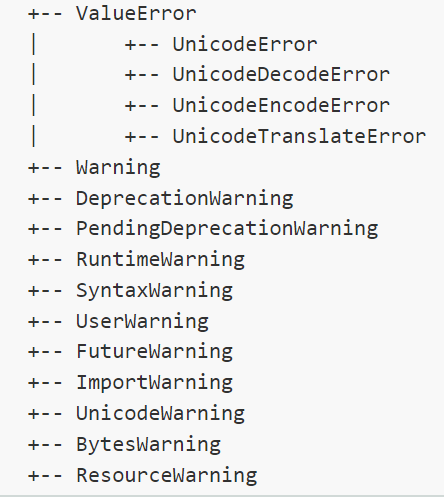
Чтение и запись в файлы это особенно хорошо.

**B8.2. Тонкости обработки исключений. Собственные классы исключений.**



Забавная шутка. Оценю сухость от 0 до 10.

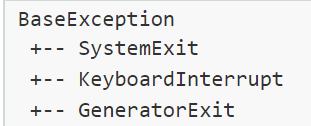
Попробуем написать свой класс исключения. Но сначала какая-то схема:

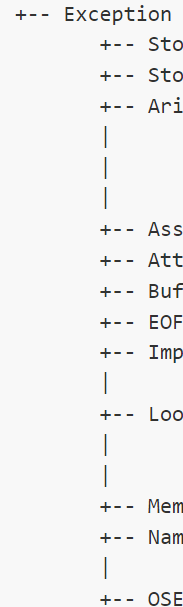
Эта система — **дерево стандартных исключений**.

Вся эта куча это стандартные исключения, которые наследуются от класса **BaseException.**

**SystemExit, KeyboardInterrupt**, **GeneratorExit** это исключения, которые нельзя отловить. Причина в том, что их возникновение не зависит от выполнения программы.



А вот все остальное, что наследуется от класса **Exception,** можно**.**



Хотя некоторые из них и возникают крайней редко. Важно понять, что в блоке except можно ловить не только сам класс, но и его родителя:

